AFRL-IF-RS-TR-2005-301
Final Technical Report
August 2005

CORE SEMANTICS FOR PUBLIC ONTOLOGIES

University of West Florida

Sponsored by
Defense Advanced Research Projects Agency
DARPA Order No. K537

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

STINFO FINAL REPORT

AFRL-IF-RS-TR-2005-301 has been reviewed and is approved for publication

APPROVED: /s/

JOSEPH A. CAROZZONI
Project Engineer

FOR THE DIRECTOR: /s/

JAMES W. CUSACK, Chief
Information Systems Division
Information Directorate

REPORT DOCUMENTATION PAGE	*Form Approved OMB No. 074-0188*

1. AGENCY USE ONLY (Leave blank)	**2. REPORT DATE** AUGUST 2005	**3. REPORT TYPE AND DATES COVERED** Final Aug 00 – Dec 04

4. TITLE AND SUBTITLE CORE SEMANTICS FOR PUBLIC ONTOLOGIES	**5. FUNDING NUMBERS** C - F30602-00-2-0577 PE - 62702E PR - DAML TA - 00 WU - 16
6. AUTHOR(S) Niranjan Suni	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) University of West Florida Institute for Human & Machine Cognition 40 South Alcaniz Street Pensacola Florida 32501	**8. PERFORMING ORGANIZATION REPORT NUMBER** N/A
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) Defense Advanced Research Projects Agency 3701 North Fairfax Drive Arlington Virginia 22203-1714 AFRL/IFSB 525 Brooks Road Rome New York 13441-4505	**10. SPONSORING / MONITORING AGENCY REPORT NUMBER** AFRL-IF-RS-TR-2005-301

11. SUPPLEMENTARY NOTES

AFRL Project Engineer: Joseph A. Carozzoni/IFSB/(315) 330-7796/ Joseph.Carozzoni@rl.af.mil

12a. DISTRIBUTION / AVAILABILITY STATEMENT APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.	**12b. DISTRIBUTION CODE**

13. ABSTRACT *(Maximum 200 Words)*

The World Wide Web (WWW) contains a large amount of information which is currently being represented using the Hypertext Markup Language (HTML) and the Extensible Markup Language (XML). However, HTML and XML have a limited capability to describe the relationships (schemas or ontologies) with respect to objects. The DARPA Agent Markup Language (DAML) through the use of ontologies provides a very powerful way to describe objects and their relationships to other objects. DAML is an extension to XML and the Resource Description Framework (RDF). This effort focused on development of semantic transfer protocols for exchange of information and communication between semantically competent agents. It also developed an extended theory semantically expressive description languages used by and shared between agents.

14. SUBJECT TERMS DAML, W3, RDF, OIL, Software Agents	**15. NUMBER OF PAGES** 23
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	**18. SECURITY CLASSIFICATION OF THIS PAGE** UNCLASSIFIED	**19. SECURITY CLASSIFICATION OF ABSTRACT** UNCLASSIFIED	**20. LIMITATION OF ABSTRACT** UL

Table of Contents

INTRODUCTION **1**

CHAPTER 1 STRONG MOBILITY AND FINE-GRAINED RESOURCE CONTROL IN NOMADS1 .... **3**

1. INTRODUCTION **3**

2. NOMADS CAPABILITIES **4**

2.1 STRONG MOBILITY 5
2.2 SAFE EXECUTION 6
2.3 MISCELLANEOUS FEATURES 7

3. AROMA VIRTUAL MACHINE **7**

3.1 CAPTURING EXECUTION STATE 8
3.2 ENFORCING RESOURCE LIMITS 9

4. OASIS EXECUTION ENVIRONMENT **10**

5. PERFORMANCE **12**

5.1 AGENT MOBILITY PERFORMANCE[3] 12
5.2 RESOURCE CONTROL PERFORMANCE 13

6. CONCLUSIONS **14**

REFERENCES **15**

CHAPTER 2 STATE CAPTURE AND RESOURCE CONTROL FOR JAVA: THE DESIGN AND IMPLEMENTATION OF THE AROMA VIRTUAL MACHINE **16**

1. INTRODUCTION **16**

List of Figures

FIGURE 1. THE OASIS EXECUTION ENVIRONMENT 11

List of Tables

TABLE 1. COMPARISON OF JUMP AGENT PERFORMANCE 13
TABLE 2. NOMADS RESOURCE CONTROL PERFORMANCE RESULTS 14

INTRODUCTION

NOMADS is a Java-based mobile agent system. It is an infrastructure that allows mobile agents to be developed and deployed. Mobile agents are programs that can behave autonomously and can move from one system to another over a network. Mobile agents are a useful approach to solving a number of problems. Agents in general are very good for delegating tasks. Agents can be written to accept high-level goals from a user describing a task to be completed and satisfy the goals based on their prior knowledge, experiences, and preferences of the user. Along these lines, agents can be very useful in gathering, filtering, and highlighting information, which makes them useful for command and control scenarios. Agents can also be very useful for monitoring state waiting for changes to occur over long periods of time. In general, agents are good for automating many types of repetitive tasks.

Mobile agents extend the agent metaphor by adding a new capability - the ability for an agent to move from one system to another across a network. This movement may be user-directed or self-directed (or a combination). Mobility allows agents to be used in new scenarios. For example, mobile agents can be used to gather information from various sites by actually moving themselves to the remote sites. By moving from one host to another, mobile agents can reduce network bandwidth usage, reduce communication and interaction latency, and allow for customized operations to be performed on remote systems.

A key feature of mobile agents is the support for disconnected operation, which allows a system to transmit an agent to a remote system and then disconnect from the network. The agent can remain on the remote system (or even travel to other systems) completing its task. Eventually, when the original system reconnects to the network, the agent can go back and report any results.

Another key feature of mobile agents is the ability to dynamically download new capabilities to systems. Mobile agents can be sent to systems that have already been deployed out in the field. These agents can add new behavior to the system, customize the system, or use the system in new ways that were not envisioned when the system was deployed. For example, suppose that sensors of various kinds have been deployed. If the current data-reporting capabilities are inadequate to accommodate an unusual situation and the sensors were capable of receiving mobile agents, a mobile agent could be sent to the sensor with a new algorithm to gather data as needed and report back to the user. Agents could also be used to upgrade or reprogram the capabilities of the sensors in general.

The NOMADS mobile agent system uses the AromaVM, which gives NOMADS a few key enhancements over other systems. For example, NOMADS can support strong (or transparent) mobility of agents because the AromaVM allows the capture of execution state. Strong mobility makes mobile agents easier to program by providing a paradigm that is easier to understand and easier to use. NOMADS also supports forced mobility, which allows the system to move agents between hosts potentially even without the agents realizing the relocation. Such forced mobility of agents is very useful for load-balancing and evacuation.

Finally, the AromaVM allows resource limits to be placed on processes. NOMADS uses this capability to limit resources that an agent uses on a system. This is a very important component of providing secure execution environments. Without secure execution environments, it would be dangerous to run mobile agents that originate from untrusted places. Even if only trusted agents are allowed, it is quite possible that an agent has been tampered with during the process of traveling. Without a secure execution environment, the system would be open to several kinds of attacks. In particular, without the kind of resource limits provided by the AromaVM, it would be easy to launch denial of service attacks that use up valuable system and network resources.

Chapter 1 Strong Mobility and Fine-Grained Resource Control in NOMADS1

1. Introduction

Mobile agent systems may be classified into two categories: those that support strong mobility and those that do not. Systems that provide strong mobility are able to capture and transfer the full execution state of the migrating agent. Systems that support only weak mobility do not transfer the execution state but rather restart execution of the agent on the remote system.

The Mobile Agent List [7] identifies over 60 mobile agent systems with the overwhelming majority being Java-based. Examples of well-known Java-based systems include Aglets [10], D'Agents [6], Voyager [14], and Concordia [12]. Because Sun's Java Virtual Machine (VM) does not allow execution state capture, very few of the Java-based mobile agent systems provide strong mobility. Those that do fall into two categories: systems using a modified Java VM and systems using a preprocessor approach.

Sumatra [1] and Ara [11,15] are two systems that use a modified Java VM to provide strong mobility. One problem with this approach is that the modified VM cannot be redistributed due to licensing constraints.[2] A second problem is that both of these systems were based on Java Developer Kit (JDK) 1.0.2 VM, which did not use native threads. Since JDK 1.2 and JDK 1.3 VMs rely on native threads, modifying the newer VMs to capture execution state would be more difficult.

The WASP system [5] uses a preprocessor approach to provide strong mobility. The advantage of the preprocessor approach is the ability to work with the standard JDK VM. However, one of the disadvantages of the preprocessor approach is the overhead introduced by the additional code added by the instrumentation process. Another disadvantage is that capturing execution state of multiple threads requires that each thread periodically poll the other threads to see if any of them have requested a move operation. This polling adds additional overhead and complicates the task of writing agents with multiple threads.

Our approach in NOMADS was to develop a custom VM (called Aroma) that has the ability to capture thread execution state. Since the Aroma VM does not use any source code from Sun's VM implementation, there are no licensing constraints on redistributing the NOMADS system. Implementing the state capture in the VM gives us the capability to transparently handle multiple threads and to support additional kinds of mobility such as forced mobility.

Another important feature of the NOMADS system is dynamic resource control. Early versions of Java relied on the sandbox model to protect mobile code from accessing dangerous methods. In contrast, the security model in the Java 2 release is permission-based. Unlike the previous "all or nothing" approach, Java applets and applications can be given varying amounts of access to system resources based upon policies. Because these policies are external to the programs, the policies can be created and modified as appropriate by a developer, system or network administrator, the end user, or even a Java program. The policy-based approach is a major advance, however current policies and underlying Java mechanisms do not address the problem of resource control. For example, while it may be possible to prevent a Java program from writing to any directory except /tmp (an access control issue), once the program is given permission to write to the /tmp directory, no further restrictions are placed on the program's I/O (a resource control issue). As another example, there is no current Java policy or mechanism available to limit the amount of disk space the program may use or to control the rate at which the program is allowed to read and write from the disk drive.

One attempt to provide resource control in Java is JRes [4] which provides CPU, network, and memory control. JRes uses a preprocessor to instrument code, allowing it to take into account object allocations for memory control. A second attempt [9] uses a modified Java VM to provide CPU resource control and scheduling. The Ajanta mobile agent system [16] takes a different approach by using proxies between the Java agents and resources to account for and limit the resources used by the agents. In the case of NOMADS, the Aroma VM enforces all of the resource controls and therefore does not rely on any preprocessing or special API at the Java code level. Agents simply use the standard Java platform API. Also, the overhead introduced by the resource control code in NOMADS is very low since the resource control is implemented in native code inside the VM (see performance results in section 5).

The rest of this paper is organized as follows. The next section describes the capabilities of the NOMADS system and some potential applications. Section three describes the implementation of the Aroma VM and in particular the implementation of the state capture and resource control mechanisms. Section four describes the Oasis agent execution environment. Section five presents our performance results to date and compares NOMADS with other mobile agent systems. Finally, section six concludes the paper and briefly discusses our plans for future work.

2. NOMADS Capabilities

The NOMADS environment is composed of two parts: an agent execution environment called Oasis and the Aroma VM. The combination of Oasis and Aroma provides two key enhancements over today's Java agent environments:

1. Strong mobility, the ability to capture and transfer the full execution state of mobile agents. This allows agents to be moved "anytime" at the demand of the server or the agent rather than just at specific pre-determined points.
2. Safe execution, the ability to control the resources consumed by agents thereby facilitating guarantees of quality of service while protecting against denial of service attacks. Adding these resource control capabilities to the access control mechanisms already provided by the new Java 2 security model allows mobile agents to be deployed with greater confidence in open environments.

2.1 Strong Mobility

Strong mobility simplifies the task of the agent programmer. Since strong mobility preserves the execution state, mobility can be invoked by the agent simply by calling the appropriate API anywhere in the code. The code fragment below shows a simple NOMADS agent that displays a message on one system, moves to another system, and displays another message.

```
public class Visitor extends Agent

{

    public static void main (String[] args)

    {

        System.out.println (“On source”);

        go (“<destination host>”);

        System.out.println (“On destination”);

    }
}
```

The following code fragment shows an Aglets agent that performs the same task. Note that in this case a Boolean variable has to be introduced external to the run() method in order to store state information. This added complexity is not peculiar to Aglets but to any mobile agent system that does not provide strong mobility.

```
public class Visitor extends Aglet

{

    public void run()

    {

        if (_theRemote) {

            System.out.println (“On destination”);

        }

        else {
            System.out.println (“On source”);

            _theRemote = true;

            dispatch (destination);

        }

    }

    protected Boolean _theRemote = false;
}
```

Strong mobility is also vital for situations in which there are long-running or long-lived agents and, for reasons external to the agents, they need to suddenly move or be moved from one host to another. In principle, such a transparent mechanism would allow the agents to continue running without any loss of their ongoing computation and, depending on circumstances, the agents need not even be aware of the fact that they have been moved (e.g., in forced mobility situations). Such an approach will be useful in building distributed systems with complex load balancing requirements. The same mechanism could also be used to replicate agents without their explicit knowledge. This would allow the support system to replicate agents and execute them on different hosts for safety, redundancy, performance, or other reasons (e.g., isolating and observing malicious agents without their knowledge).

Exploiting Java’s byte code approach, NOMADS allows the execution state of an agent to be captured on one host of one architecture (say an Intel x86 running Windows NT) and restored on another host with a different architecture (such as a Sun SPARC running Solaris). While it is possible to achieve some measure of transparent persistence by techniques such as having a special class loader insert read and write barriers into the source code before execution, such an approach poses many problems [8]. First, the transformed byte codes could not be reused outside of a particular persistence framework, defeating the Java platform goal of code portability. Second, such an approach would not be applicable to the core classes, which cannot be loaded by this mechanism. Third, the code transformations would be exposed to debuggers, performance monitoring tools, the reflection system, and so forth, compromising the goal of complete transparency.

We note that the current version of NOMADS does not provide any mechanism to transparently access resources independent of agent mobility. Section 6 briefly describes our current efforts for transparently redirecting network and disk resources.

2.2 Safe Execution

Mechanisms for monitoring and controlling agent use of host resources are important for three reasons [13]. First, it is essential that access to critical host resources such as the hard disk be denied to unauthorized agents. Second, the use of resources to which access has been granted must be kept within reasonable bounds, making it easier to provide a specific quality-of-service for each agent. Denial-of-service conditions resulting from a poorly programmed or malicious agent's overuse of critical resources are impossible to detect and interrupt without monitoring and control mechanisms for individual agents. Third, tracking of resource usage enables accounting and billing mechanisms that hosts may use to calculate charges for resident agents.

The Aroma VM provides flexible and dynamic resource control for disk and network resources. Using Aroma, it is possible to limit both the rate and the quantity of resources that each agent is allowed to use. Resource limits that may be enforced include disk and network read and write rates, total number of bytes read and written to disk and network, and disk space. Note that disk space is different from disk bytes written because of seek operations that may be performed on disk files. The rate limits are expressed in bytes/sec whereas the quantity limits are expressed in bytes.

Once an agent is authenticated, a policy file specifies the initial set of limits that should be enforced by Aroma. These limits are dynamically changeable through the Oasis administration program (discussed in section 4). Dynamically changing resource limits is also beneficial to prioritizing agents that are running on a host. In a related project, we are working on a high-level policy-based agent management system that resides on top of the low-level enforcement capabilities of the Aroma VM [2,3].

2.3 Miscellaneous Features

NOMADS provides several other features to support agents and the tasks agents may need to perform. A low-level messaging API is provided to allow an agent to send a message to another agent. Agents are assigned UUIDs upon creation and the UUIDs are used to address agents when sending messages. Agents may also use alternate names for the convenience of people or other agents. A directory service maps agent names to their UUIDs. Agents may use the Java Platform API for creating new threads, synchronizing between threads, accessing files, networks, and for performing I/O.

3. Aroma Virtual Machine

The Aroma VM is a Java-compatible VM designed and implemented with the specific requirements of strong mobility and safe execution. The primary goals for Aroma were to support:

1. Capture of the execution state of a single Java thread, thread group, or all threads (complete process) in the VM;
2. Capture of the execution state at fine levels of granularity (ideally, between any two Java byte code instructions);
3. Capture of the execution state as transparently to the Java code executing in the VM as possible;
4. Cross-platform compatibility for the execution state information;
5. Flexibility in how much information is captured (in particular whether to include the definitions of Java classes);

6. Easy portability to a variety of platforms (at least Win32 and various UNIX/Linux platforms);
7. Flexible usage in different contexts and inside different applications;
8. Enforcement of fine-grained and dynamically changing limits of access to resources such as the CPU, memory, disk, network, and GUI.

In the current version of Aroma, the VM can only capture the execution state of all threads rather than just a designated subset. Also, at the present time, only the disk and network resource limits have been implemented. These limitations will be addressed in future versions of Aroma.

The Aroma VM is implemented in C++ and consists of two parts: the VM library and a native code library. The VM library can be linked to other application programs. Currently, two programs use the VM library – avm (a simple wrapper program that is similar to the java executable) and oasis (the agent execution environment). The VM library consists of approximately 40,000 lines of C++ code. The native code library is dynamically loaded by the VM library and implements the native methods in the Java API. Both the VM and the native code libraries have been ported to Win32, Solaris (on SPARC) and Linux (on x86) platforms. In principle, the Aroma VM should be portable to any platform that supports ANSI C++, POSIX or Win32 threads, and POSIX style calls for file and socket I/O. We plan to port the Aroma VM to WinCE-based platforms, and expect that a port to Macintosh OS X when it is available will be straightforward as well.

3.1 Capturing Execution State

Aroma is capable of capturing the execution state of all threads running inside the VM. This state capture may be initiated by either a thread running inside or outside the VM. The former is useful when the agent requests an operation that needs the execution state to be captured. The latter is useful when the system wants the execution state to be captured (for example, to implement forced mobility).

For several reasons, we chose to map each Java thread to a separate native operating system thread. The other alternatives were to develop our own threads package (which would be platform specific and difficult to port) or use an existing threads package (which may or may not be available on different platforms). Also, mapping to native threads allows the VM to take advantage of the presence of multiple CPUs. Therefore, if a VM has two Java threads running (JT1 and JT2), then there are two native threads (NT1 and NT2) that correspond to JT1 and JT2. If the execution state of the VM is captured at this point and restored later (possibly on a new host), then two new native threads will be created (NT3 and NT4) to correspond to the two Java threads JT1 and JT2.

However, mapping Java threads to native threads complicates the mechanism of capturing the execution state. This is because when one Java thread (or some external thread) requests a state capture, the other concurrently running threads may be in many different states. For example, other Java threads could be blocked trying to enter a monitor, waiting on a condition variable, sleeping, suspended, or executing native code. We wanted as few restrictions as possible on when a thread's state may be captured so that we can support capturing execution state at fine levels of granularity. Therefore, the implementation of monitors was carefully designed to accommodate state capture. For example, if a Java thread is blocked trying to enter a monitor, then there is a corresponding native thread that is also blocked on some IPC primitive. If at this point the execution state is captured and restored later (possibly on a different system and of a different architecture), a new native thread must resume in the same blocked state that the original native thread was in when the state was captured. To support this capability, the monitors were designed in such a way that native threads blocked in monitors could be terminated and new native threads could take their "place" in the monitor at a later point in time. As an example, consider a native thread NT1 on host H1 that represents a Java thread JT1. NT1 could be blocked because JT1 was trying to enter a monitor. The VM will allow another thread to capture the execution state at such a time and when the state is restored later, a new native thread NT2 (on possibly a new host H2) will be created to represent JT1. Furthermore, NT2 will continue to be blocked in the monitor in the same state as NT1.

Another requirement is the need to support multiple platforms. In particular, to support capturing the execution state on one platform (such as Win32) and restoring the state on a different platform (such as Solaris SPARC). The Java byte code format ensures that the definitions of the classes are platform independent so transferring the code is not an issue. For transferring the execution state, the Aroma VM assumes that the word size is always 32-bits and that the floating-point representations are the same. With these assumptions, the only other major issue is transferring state between little-endian and big-endian systems. The Aroma VM writes a parameter as part of the state information indicating whether the source platform was big- or little-endian. The destination platform is responsible for byte-swapping values in the execution state if necessary.

One limitation is that if any of the Java threads are executing native code (for example, by invoking a native method), then the VM will wait for the threads to finish their native code before initiating the state capture. This limitation is necessary because the VM does not have access to the native code execution stack.

3.2 Enforcing Resource Limits

The native code library is responsible for implementing the enforcement of resource limits. The current version is capable of enforcing disk and network limits. The limits may be grouped into three categories: rate limits, quantity limits, and space limits. Rate limits allow the read and write rates of any program to be limited. For example, the disk read rate could be limited to 100 KB/s. Similarly, the network write rate could be limited to 50 KB/s. The rate limits ensure that an agent does not exceed the specified rate for any input and output operations. For example, if a network write rate of 50 KB/s was in effect and a thread tried to write at a higher rate, the thread would be slowed down until it does not exceed the write rate limit.

Quantity limits allow the total bytes read or written to be limited. For example, the disk write quantity could be limited to 3 MB. Similarly, the network read quantity could be limited to 1 MB. If an agent tried to read or write more data than allowed by the limit, the thread performing the operation would get an IOException.

The last category of limits is the space limit, which applies only to disk space. Again, if an agent tries to use more space than allowed by the disk space limit, then the VM would throw an IOException. Note that the disk space limit is different from the disk write quantity limit. If an agent has written 10 MB of data, it need not be the case that the agent has used up 10 MB of disk space because the agent could have written over the same file(s) or erased some of the files that it had written.

To enforce the quantity limits, the native code library maintains four counters for the number of bytes read and written to the network and the disk. For every read or write operation, the library checks whether performing the operation would allow the agent to exceed a limit. If so, the library returns an exception to the agent. Otherwise, the appropriate counter is incremented and the operation is allowed to proceed. To enforce the disk space limit, the library performs a similar computation except that seek operations and file deletions are taken into consideration. Again, if an operation would allow the agent to exceed the disk space limit, the library returns an exception and does not complete the operation.

To enforce the rate limits, the library maintains four additional counters for the number of bytes read and written to the network and the disk and four time variables, which record the time when the first operation was performed. Before an operation is allowed, the library divides the number of bytes by the elapsed time to check if the agent is above the rate limit. If so, the library puts the thread to sleep until such time that the agent is within the rate limit. Then, the library computes how many bytes may be read or written by the agent in a 100ms interval. If the operation requested by the agent is less than what is allowed in a 100ms interval, the library simply completes the operation and returns (after updating the counter). Otherwise, the library divides the operation into sub-operations and performs them in each interval. After an operation is performed, the library sleeps until the interval finishes. For example, if an agent requested a write of 10 KB and the write rate limit was 5 KB/s, then the number of bytes that the agent is allowed to write in a 100ms interval is 512 bytes. Therefore, the library would loop 20 times, each time writing 512 bytes and then sleeping for the remainder of the 100ms interval. One final point to make is that if a rate limit is changed then the counter and the timer are reset. This reset is necessary to make sure that the rate limit is an instantaneous limit as opposed to an average limit.

4. Oasis Execution Environment

Oasis is an agent execution environment that embeds the Aroma VM. It is divided into two independent programs: a front-end interaction and administration program and a back-end execution environment. Figure 1 shows the major components of Oasis. The

Oasis Console program may be used to interact with agents running within the execution environment. The console program is also used to perform administrative tasks such as creating accounts, setting policies, and changing resource limits.

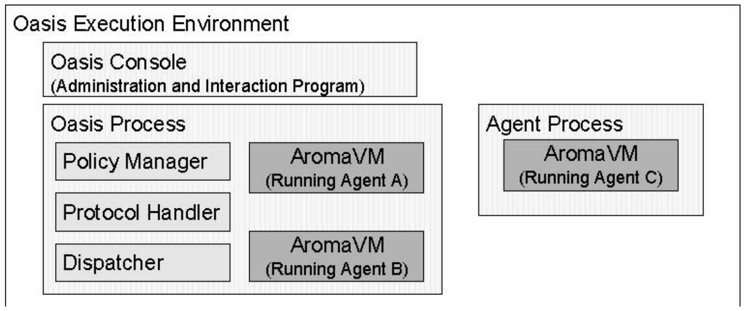

Figure 1. The Oasis Execution Environment

The Oasis process is the execution environment for agents. Among other things, it contains instances of the Aroma VM for running agents, a Policy Manager, one or more Protocol Handlers, and a Dispatcher. Each agent executes in a separate instance of the Aroma VM, and can use multiple threads if desired. In the normal case, all the instances of the Aroma VM are inside the same Oasis process. However, it is possible to execute agents in separate isolated agent processes. One advantage of using a separate process is that the Oasis process itself can be stopped and restarted without stopping the agents.

The policy manager is a major component of the execution environment. It is responsible for establishing security and resource control policies for all agents. Policies address authentication and agent transfer, execution control, access control, and resource usage. As Figure 1 shows, a user or administrator may interact with the Oasis environment through a separate administration process that allows the user to examine and change various resource limits.

Oasis can support multiple protocol handlers. A protocol handler is responsible for transferring the state of an agent from one Oasis to another. The default protocol handler in Oasis implements a custom agent transfer protocol but new protocol handlers can be written to support other (standard) protocols. The default protocol handler is capable of compressing and decompressing agent state information on the fly while an agent is being transferred.

One important design choice was to run each agent within a separate instance of the Aroma VM. Such a design has both advantages and disadvantages. The advantage is that resource accounting and control is simplified. The disadvantage is increased overhead. We are working on the possibility of sharing class definitions between multiple Aroma VMs which should reduce the overhead significantly.

One of the capabilities provided by Oasis is the dynamic adjustment of resource limits of agents. This causes a potential problem for certain kinds of resources when resource limits are lowered below the threshold already consumed by an agent. For example, an agent may have already used 10 MB of disk space and the resource limit might be reduced to 8 MB. The current implementation does not attempt to reclaim the 2 MB of disk space back from the agent. Instead, any future requests for disk space simply fail until the agent's disk usage drops below 8 MB. In the future, we would like to explore a mechanism to notify an agent about the change in resource limits (using a callback function) and allow the agent a fixed amount of time for the agent to comply with the changed limits or perhaps to negotiate some compromise. If the agent does not comply then Oasis has the option of terminating the agent or transmitting the agent back to its home or to some other designated location.

5. Performance

This section describes some initial performance results of NOMADS. The performance measurements are divided into two categories: performance of agent mobility and performance of resource limits. We have yet not collected any data but superficial comparisons indicate that the Aroma VM is significantly slower than the Sun's Java VM. Once we have completed implementation of necessary features, we will focus more attention on performance optimization and testing.

5.1 Agent Mobility Performance[3]

We compared the mobility performance of NOMADS with three other Java-based systems including Aglets 1.1b2 [10], Concordia 1.14 [12], and Voyager 3.2 [14]. For each platform, we wrote a simple mobile agent that carried a payload of a specified size. The objective was to measure the round-trip time of each agent (i.e., the time taken for the agent carrying the payload to move from the source host to the destination and return back to the source. In our experiments, the independent variables were the system type and the agent payload size (0, 16 KB, and 64 KB). In the case of NOMADS, an additional independent variable was compression of agent state information. The dependent variable was the round-trip time for the agent.

The equipment used for the experiments were two Pentium III systems operating at 650 MHz with 256 MB RAM each running Windows NT 4.0 with Service Pack 6. The systems were on an isolated 100 Mbps Fast Ethernet network. The results are summarized in the table 1 below. The times reported for NOMADS-C are the times for NOMADS with compression enabled for agent transfer. All times are in milliseconds.

Table 1. Comparison of Jump Agent Performance

	NOMADS	NOMADS-C	Aglets	Concordia	Voyager
0 KB	333.5	443.8	90.6	138.5	115.6
16 KB	337.4	446.7	100.8	147.7	124.7
64 KB	341.6	448.7	144.8	182.3	169.3

The results show that the performance of NOMADS ranges from 1.87 to 3.7 times slower than the other systems. NOMADS is slower because of the additional execution state information that is being transferred. The relative performance of NOMADS is better when the agents are larger (or are carrying a large payload). Another interesting result is the tradeoff between CPU-time and transfer time with and without compression. On a 100 Mbps network, enabling compression actually decreases performance because more time is spent in the compression and decompression phase than the actual transfer phase. We expect the compression feature would be more useful on a slower connection, such as one implemented on a wireless network. Finally, it is also interesting to note that the performance of NOMADS is virtually unchanged irrespective of the payload size. This can be explained by the fact that the size of the payload is insignificant when compared to the size of the agent state (which is several hundred KB).

We expect the performance of NOMADS to improve significantly after optimization. For example, currently, the state information transferred by NOMADS includes all the Java class definitions. We plan to optimize the system by not transferring the class definitions of those classes that are already available on the remote system. We also plan to support capturing the execution state of individual threads or thread groups, which should significantly improve performance.

5.2 Resource Control Performance

To report on the performance of resource control in NOMADS, we measured the overhead imposed by the resource control code on the overall performance of I/O operations. We took four different performance measurements:

The Java VM from Sun Microsystems,
The Aroma VM with no resource control code
The Aroma VM with resource control code but no resource limit in place
The Aroma VM with resource control code and a very high resource limit in place (i.e., one that will never be exceeded by the agent).

The measurements were taken using an agent that continuously writes data to the network. Two variants of the agent were used, one that wrote the data in 10 KB blocks and the other that wrote the file in 64 KB blocks. In each case, we measured the I/O rate in bytes per millisecond.

Table 2. NOMADS Resource Control Performance Results

	10 KB	64 KB
Java VM	9532	9903
Aroma VM (with no resource control code)	8772	9650
Aroma VM (with code but no limit)	8746	9656
Aroma VM (with code but very high limit)	8702	9655

The results show that the overhead imposed by the resource control code is minimal. Although we have not measured the overhead of the disk resource limits, we expect them to be very similar since the mechanism for enforcing the limits is the same.

6. Conclusions

We have described our motivations for developing a mobile agent system that provides strong mobility and safe execution. We have also described the initial design and implementation of the Aroma VM and the Oasis agent execution environment. Initial performance results are promising. The speed of agent transfer using the current unoptimized NOMADS code ranges only from 1.87 to 3.7 times slower than the weak mobility systems evaluated and we have several ideas for significantly increasing performance. The overhead for resource control in our experiment was insignificant.

To date, both the Aroma VM and Oasis have been ported to Windows NT, Solaris (on SPARC), and Linux (on x86). The Aroma VM is currently JDK 1.2 compatible with some limitations and missing features, the major omission being support for AWT. The AWT implementation affords opportunities for evaluating resource management mechanisms for graphical resources. NOMADS may be downloaded at no cost for educational and research use from http://www.coginst.uwf.edu/nomads.

Work is currently underway on optimizing the transfer of execution state information, capturing execution state of individual threads and thread groups, implementing transparent redirection for both disk and network resources, and resource controls for CPU and memory. We are also working on sophisticated high-level agent management tools [2,3] that build on top of the resource control capabilities of NOMADS.

References

1. Acharya, A., Ragnganathan, M., & Saltz, J. Sumatra: A language for resource-aware mobile programs. In J. Vitek & C. Tschudin (Ed.), Mobile Object Systems. Springer-Verlag.
2. Bradshaw, J. M., Greaves, M., Holmback, H., Jansen, W., Karygiannis, T., Silverman, B., Suri, N., & Wong, A. Agents for the masses: Is it possible to make development of sophisticated agents simple enough to be practical? IEEE Intelligent Systems(March-April), 53-63.
3. Bradshaw, J. M., Cranfill, R., Greaves, M., Holmback, H., Jansen, W., Jeffers, R., Karygiannis, T., Kerstetter, M., Suri, N. & Wong, A. Policy-based management of agents and domains, submitted for publication.
4. Czajkowki, G., & von Eicken, T. JRes: A resource accounting interface for Java. Proceedings of the 1998 ACM OOPSLA Conference. Vancouver, B.C., Canada.
5. Fünfrocken, S. Transparent migration of Java-based mobile agents: Capturing and reestablishing the state of Java programs. In K. Rothermel & F. Hohl (Ed.), Mobile Agents: Proceedings of the Second International Workshop (MA 98). Springer-Verlag.
6. Gray, R. S. Agent Tcl: A flexible and secure mobile-agent system. Proceedings of the 1996 Tcl/Tk Workshop, (pp. 9-23).
7. Hohl, F. The Mobile Agent List. http://ncstrl.informatik.uni-stuttgart.de/ipvr/vs/projekte/ mole/mal/mal.html
8. Jordan, M., & Atkinson, M. Orthogonal persistence for Java—A mid-term report. Sun Microsystems Laboratories.
9. Lal, M. & Pandey, R. CPU Resource Control for Mobile Programs. Proceedings of the First International Symposium on Agent Systems and Applications and the Third International Symposium on Mobile Agents (ASA/MA'99). IEEE Computer Society Press.
10. Lange, D. B., & Oshima, M. Programming and Deploying Java Mobile Agents with Aglets. Reading, MA: Addison-Wesley.
11. Maurer, J. Porting the Java runtime system to the Ara platform for mobile agents. Diploma Thesis, University of Kaiserslautern.
12. Mitsubishi. Concordia htttp://www.meitca.com/HSL/Projects/Concordia/ whatsnew.htm.
13. Neuenhofen, K. A., & Thompson, M. Contemplations on a secure marketplace for mobile Java agents. K. P. Sycara & M. Wooldridge (Ed.), Proceedings of Autonomous Agents 98, . Minneapolis, MN, , New York: ACM Press.
14. ObjectSpace. ObjectSpace Voyager http://www.objectspace.com/products/voyager.
15. Peine, H., & Stolpmann, T. The architecture of the Ara platform fro mobile agents. In K. Rothernel & R. Popescu-Zeletin (Ed.), Proceedings of the First International Workshop on Mobile Agents (MA 97). Springer-Verlag.
16. Tripathi, A. & Karnik, N. Protected Resource Access for Mobile Agent-based Distributed Computing. In Proceedings of the ICPP Workshop on Wireless Networking and Mobile Computing, Minneapolis, August 1998.

Chapter 2 State Capture and Resource Control for Java: The Design and Implementation of the Aroma Virtual Machine

1. Introduction

Although Java is currently riding a rising wave of popularity, current versions fail to address many of the unique challenges posed by the new generation of distributed applications. In particular the advent of peer-to-peer computing models and the proliferation of software agents motivates various requirements that go beyond the capabilities of current Java Virtual Machines:

- Full state capture. To support check-pointing and load balancing, the Virtual Machine (VM) must be able to capture its complete state including all threads, objects, and classes in the heap. To support requirements for strong "anytime" mobility and forced migration (such as when a host is about to go offline), the VM must be able to support asynchronous requests to capture execution state for a thread or thread group.

- Dynamic access and resource control. The security model in Java 2 [4] provides a fairly comprehensive model for access control but does not allow for dynamic permission revocation. Once permission is granted to a process, that permission is in effect for the lifetime of that process. Furthermore, there is no way of specifying the amount of a resource that is granted, assuring a specific quality of service for each process. For example, one would like to be able to limit the quantity of hard disk, network, or CPU usage that is available to a given process or to determine the rate at which the resource may be used. Denial-of-service conditions on a host or network resulting from code that is poorly programmed, malicious, or has been tampered with are impossible to detect and interrupt without dynamic monitoring and control mechanisms for individual processes.

- Resource accounting. Tracking of resource use, based on resource control mechanisms, enables accounting and billing mechanisms that hosts can use to calculate charges for resident programs. The same mechanisms can be used to detect patterns of resource abuse.

The Aroma VM is a Java-compatible VM that provides unique capabilities such as thread and VM state capture and dynamic, fine-grained resource control and accounting. Aroma was developed as part of a research project on mobile agent systems and distributed systems. The overall goal was to develop a VM for research use that would be simple, flexible, and portable. Therefore, Aroma tries to minimize dependence on operating systems features and avoids platform-specific assembly code. Aroma is currently being used as part of the NOMADS mobile agent system and in the WYA ("While You're

Away”) distributed system for load balancing and utilizing idle workstations.

Aroma currently provides mechanisms to capture the state of individual threads or the complete VM. Individual thread state capture examines each of the stack frames in the method stack of the thread and saves all relevant values (such as the program counter, the local variables, and the operand stack) as well as all reachable objects. Full VM state capture saves the state of all threads in the VM as well as all loaded classes. State capture may be initiated synchronously by a thread or asynchronously by an external request.

Java threads in Aroma have a one-to-one mapping to native operating system threads. The primary reason for mapping directly to native threads was to not have a platform-specific user-level threads package. However, mapping to native threads complicates the task of capturing a Java thread’s state. In particular, asynchronous requests to capture thread state are harder to support because threads may be in one of many states (such as running, blocked, waiting, sleeping, or suspended) when the state capture is requested.

Several components of Aroma were carefully designed to support state capture of asynchronous Java threads. The Java threads and their corresponding native threads are decoupled so that the native thread’s stack stays constant while allowing the Java thread’s stack to change with method invocations and returns. Aroma also uses special stack frames on the Java thread stack to handle situations where Java code and C code might be interleaved (such as a Java method instantiating an object which requires native code to load the class which might again require a Java method to be invoked to initialize the class). Finally, the monitors in Aroma were carefully designed to be portable so that the state of Java threads can be captured even if they are blocked trying to enter a monitor or waiting on a condition variable.

Aroma currently provides a comprehensive set of resource controls for CPU, disk, and network. The resource control mechanisms allow limits to be placed on both the rate and quantity of resources used by Java threads. Rate limits include CPU usage, disk read rate, disk write rate, network read rate, and network write rate. Rate limits for I/O are specified in bytes/millisecond. Quantity limits include disk space, total bytes written to disk, total bytes read from the disk, total bytes written to the network, and total bytes read from the network. Quantity limits are specified in bytes.

CPU resource control was designed to support two alternative means of expressing the resource limits. The first alternative is to express the limit in terms of byte codes executed per millisecond. The advantage of expressing a limit in terms of byte codes per unit time is that given the processing requirements of a thread, the thread’s execution time (or time to complete a task) may be predicted. Another advantage of expressing limits in terms of byte codes per unit time is that the limit is system and architecture independent. The second alternative is to express the limit in terms of some percentage of CPU time, expressed as a number between 0 and 100. Expressing limits as a percentage of overall CPU time on a host provides better control over resource consumption on that particular host.

Rate limits for disk and network are expressed in terms of bytes read or written per millisecond. If a rate limit is in effect, then I/O operations are transparently delayed if necessary until such time that allowing the operation would not exceed the limit. Threads performing I/O operations will not be aware of any resource limits in place unless they choose to query the VM.

Quantity limits for disk and network are expressed in terms of bytes. If a quantity limit is in effect, then the VM throws an exception when a thread requests an I/O operation that would result in the limit being exceeded.

The Aroma VM implementation is based on the Java Virtual Machine specification and does not use any source code from other licensed VM implementations. Therefore, Aroma may be distributed without any licensing constraints. Currently, Aroma is distributed in binary form as bundled with the NOMADS mobile agent system. NOMADS may be downloaded and used free of charge for non-commercial purposes from http://www.coginst.uwf.edu/nomads. We also plan to release the Aroma VM in the form of an object library that may be embedded inside other applications.

Aroma is currently JDK 1.2.2 “compatible” with some missing features such as support for AWT and Swing. Also, Aroma currently works with the Sun implementation of the Java Platform API (as distributed in the Java Runtime Environment). In the future, we also plan to support other API implementations such as the GNU Classpath project. Aroma has been ported to Win 32 on x86, Solaris on SPARC, and Linux on x86.

Currently, Aroma does not provide a Just-in-Time compiler, which significantly affects the performance of Aroma when compared with other VM implementations. In the future, we will work on integrating freely available JIT compilers (such as OpenJIT) while still retaining the unique features of Aroma. The VM does offer good state capture performance and has minimal overhead for disk and network resource controls. CPU resource control introduces an overhead of 6.6% to 6.9%. More information on the Aroma VM, the NOMADS mobile agent system, and the WYA (While You’re Away) distributed system is available on the NOMADS web site at http://www.coginst.uwf.edu/nomads.